Join the discussion @ p2p.wrox.com

wrox

Wrox Programmer to Programmer™

United States District Court
Northern District of California

Δ DEFENDANT Δ

Case No. 4:20-cv-05640-YGR

Case Title *Epic Games, Inc. v. Apple, Inc.*

Exhibit No. DX-5562

Date Entered ____________

Susan Y. Soong, Clerk

By: ____________, Deputy Clerk

Mac OS X and iOS Internals

To the Apple's Core

Jonathan Levin

DX-5562.002

MAC OS® X AND iOS INTERNALS

Mac OS® X and iOS Internals

TO THE APPLE'S CORE

Jonathan Levin

WILEY
John Wiley & Sons, Inc.

DX-5562.005

Mac OS® X and iOS Internal

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2013 by Jonathan Levin

Published by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-11805765-0
ISBN: 978-1-11822225-6 (ebk)
ISBN: 978-1-11823605-5 (ebk)
ISBN: 978-1-11826094-4 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2011945020

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Mac OS is a registered trademark of Apple, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

To Steven Paul Jobs: From Mac OS's very first incarnation, to the present one, wherein the legacy of NeXTSTEP still lives, his relationship with Apple is forever entrenched in OS X (and iOS). People focus on his effect on Apple as a company. No less of an effect, though hidden to the naked eye, is on its architecture. I resisted the pixie dust for 25 years, but he finally made me love Mac OS... Just as soon as I got my shell prompt.

— Jonathan Levin

DX-5562.007

CREDITS

ACQUISITIONS EDITOR
Mary James

SENIOR PROJECT EDITOR
Adaobi Obi Tulton

DEVELOPMENT EDITOR
Sydney Argenta

TECHNICAL EDITORS
Arie Haenel
Dwight Spivey

PRODUCTION EDITOR
Christine Mugnolo

COPY EDITORS
Paula Lowell
Nancy Rapoport

EDITORIAL MANAGER
Mary Beth Wakefield

FREELANCER EDITORIAL MANAGER
Rosemarie Graham

ASSOCIATE DIRECTOR OF MARKETING
David Mayhew

MARKETING MANAGER
Ashley Zurcher

BUSINESS MANAGER
Amy Knies

PRODUCTION MANAGER
Tim Tate

VICE PRESIDENT AND EXECUTIVE GROUP PUBLISHER
Richard Swadley

VICE PRESIDENT AND EXECUTIVE PUBLISHER
Neil Edde

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Katie Crocker

PROOFREADER
James Saturnio, Word One New York

INDEXER
Robert Swanson

COVER DESIGNER
Ryan Sneed

COVER IMAGE
© Matt Jeacock / iStockPhoto

DX-5562.008

ABOUT THE AUTHOR

JONATHAN LEVIN is a seasoned technical trainer and consultant focusing on the internals of the "Big Three" (Windows, Linux, and Mac OS) as well as their mobile derivatives (Android and iOS). Jonathan has been spreading the gospel of kernel engineering and hacking for 15 years, and has given technical talks at DefCON as well as other technical conferences. He is the founder and CTO of Technologeeks.com, a partnership of expert like-minded individuals, devoted to propagating knowledge through technical training, and solving tough technical challenges through consulting. Their areas of expertise cover real-time and other critical aspects of software architectures, system/kernel-level programming, debugging, reverse engineering, and performance optimizations.

ABOUT THE TECHNICAL EDITORS

ARIE HAENEL is a security and internals expert at NDS Ltd. (now part of Cisco). Mr. Haenel has vast experience in data and device security across the board. He holds a Bachelor of Science Engineering in Computer Science from the Jerusalem College of Technology, Israel and an MBA from the University of Poitiers, France. His hobbies include learning Talmud, judo, and solving riddles. He lives in Jerusalem, Israel.

DWIGHT SPIVEY is the author of several Mac books, including *OS X Mountain Lion Portable Genius* and *OS X Lion Portable Genius*. He is also a product manager for Konica Minolta, where he has specialized in working with Mac operating systems, applications, and hardware, as well as color and monochrome laser printers. He teaches classes on Mac usage, writes training and support materials for Konica Minolta, and is a member of the Apple Developer Program. Dwight lives on the Gulf Coast of Alabama with his beautiful wife Cindy and their four amazing children, Victoria, Devyn, Emi, and Reid. He studies theology, draws comic strips, and roots for the Auburn Tigers ("War Eagle!") in his ever-decreasing spare time.

ACKNOWLEDGMENTS

"Y'KNOW, JOHNNY," said my friend Yoav, taking a puff from his cigarette on a warm summer night in Shanghai, "Why don't *you* write a book?"

And that's how it started. It was Yoav (Yobo) Chernitz who planted the seed to write my own book, for a change, after years of reading others'. From that moment, in the Far, Middle, and US East (and the countless flights in between), the idea began to germinate, and this book took form. I had little idea it would turn into the magnum opus it has become, at times taking on a life of its own, and becoming quite the endeavor. With so many unforeseen complications and delays, it's hard to believe it is now done. I tried to illuminate the darkest reaches of this monumental edifice, to delineate them, and leave no stone unturned. Whether or not I have succeeded, you be the judge. But know, I couldn't have done it without the following people:

> Arie Haenel, my longtime friend — a natural born hacker, and no small genius. Always among my harshest critics, and an obvious choice for a technical reviewer.
>
> Moshe Kravchik — whose insights and challenging questions as the book's first reader hopefully made it a lot more readable for all those who follow.
>
> Yuval Navon — from down under in Melbourne, Australia, who has shown me that friendship knows no geographical bounds.

And last, but hardly least, to my darling Amy, who was patient enough to endure my all-too-frequent travels, more than understanding enough to support me to no end, and infinitely wise enough to constantly remind me not only of the important deadlines and obligations. I had with this book, but of the things that are truly the most important in life.

— Jonathan Levin

CONTENTS

DX-5562.013

DX-5562.014

DX-5562.015

DX-5562.016

DX-5562.017

DX-5562.018

DX-5562.019

DX-5562.020

DX-5562.021

DX-5562.022

DX-5562.023

DX-5562.025

3

On the Shoulders of Giants: OS X and iOS Technologies

By virtue of being a BSD-derived system, OS X inherits most of the kernel features that are endemic to that architecture. This includes the POSIX system calls, some BSD extensions (such as kernel queues), and BSD's Mandatory Access Control (MAC) layer.

It would be wrong, however, to classify either OS X or iOS as "yet another BSD system" like FreeBSD and its ilk. Apple builds on the BSD primitive's several elaborate constructs — first and foremost being the "sandbox" mechanism for application compartmentalization and security. In addition, OS X and iOS enhance or, in some cases, completely replace BSD components. The venerable `/etc` files, for example, traditionally used for system configuration, are entirely replaced. The standard UN*X `syslog` mechanism is augmented by the Apple System Log. New technologies such as Apple Events and FSEvents are entirely proprietary.

This chapter discusses these features and more, in depth. We first discuss the BSD-inspired APIs, and then turn our attention to the Apple-specific ones. The APIs are discussed from the user-mode perspective, including detailed examples and experiments to illustrate their usage. For the kernel perspective of these APIs, where applicable, see Chapter 14, "Advanced BSD Aspects."

BSD HEIRLOOMS

While the core of XNU is undeniably Mach, its main interface to user mode is that of BSD. OS X and iOS both offer the set of POSIX compliant system calls, as well as several BSD-specific ones. In some cases, Apple has gone several extra steps, implementing additional features, some of which have been back-ported into BSD and OpenDarwin.

DX-5562.026

sysctl

The `sysctl(8)` command is somewhat of a standardized way to access the kernel's internal state. Introduced in 4.4BSD, it can also be found on other UN*X systems (notably, Linux, where it is backed by the `/proc/sys` directories). By using this command, an administrator can directly query the value of kernel variables, providing important run-time diagnostics. In some cases, modifying the value of the variables, thereby altering the kernel's behavior, is possible. Naturally, only a fairly small subset of the kernel's vast variable base is exported in this way. Nonetheless, those variables that are made visible play key roles in recording or determining kernel functionality.

The `sysctl(8)` command wraps the `sysctl(3)` library call, which itself wraps the `__sysctl` system call (#202). The exported kernel variables are accessed by their *Management Information Base* (MIB) names. This naming convention, borrowed from the Simple Network Management Protocol (SNMP), classifies variables by namespaces.

XNU supports quite a few hard-coded namespaces, as is shown in Table 3-1.

TABLE 3-1: Predefined sysctl Namespaces

NAMESPACE	NUMBER	STORES
`debug`	5	Various debugging parameters.
`hw`	6	Hardware-related settings. Usually all read only.
`kern`	1	Generic kernel-related settings.
`machdep`	7	Machine-dependent settings. Complements the hw namespace with processor-specific features.
`net`	4	Network stack settings. Protocols are defined in their own sub-namespaces.
`vfs`	3	File system-related settings. The Virtual File system Switch is the kernel's common file system layer.
`vm`	2	Virtual memory settings.
`user`	8	Settings for user programs.

As shown in the table, namespaces are translated to an integer representation, and thus the variable can be represented as an array of integers. The library call `sysctlnametomib(3)` can translate from the textual to the integer representation, though that is often unnecessary, because `sysctlbyname(3)` can be used to look up a variable value by its name.

Each namespace may have variables defined directly in it (for example, `kern.ostype`, 1.1), or in sub-namespaces (for example, `kern.ipc.somaxconn`, 1.32.2). In both cases accessing the variable in question is possible, either by specifying its fully qualified name, or by its numeric MIB specifier. Looking up a MIB number by its name (using `sysctlnametomib(3)`) is possible, but not vice versa. Thus, one can walk the MIBs by number, but not retrieve the corresponding names.

DX-5562.027

Using `sysctl(8)` you can examine the exported values, and set those that are writable. Due to the preceding limitation, however, you cannot properly "walk" the MIBs — that is, traverse the namespaces and obtain a listing of their registered variables, as one would with SNMP's `getNext()`. The command does have an `-A` switch to list all variables, but this is done by checking a fixed list, which is defined in the `<sys/sysctl.h>` header (`CTL_NAMES` and related macros). This is not a problem with the OS X `sysctl(8)`, because Apple does rebuild it to match the kernel version. In iOS, however, Apple does not supply a binary, and the one available from Cydia (as part of the system-cmds package) misses out on iOS-specific variables.

Kernel components can register additional `sysctl` values, and even entire namespaces, on the fly. Good examples are the security namespace (used heavily by the `sandbox` kext, as discussed in this chapter) and the `appleprofile` namespace (registered by the `AppleProfileFamily` kexts — as discussed in Chapter 5, "Process Tracing and Debugging"). The kernel-level perspective of `sysctl`s are discussed in Chapter 14.

The gamut of `sysctl(3)` variables ranges from various minor debug variables to other read/write variables that control entire subsystems. For example, the kernel's little-known kdebug functionality operates entirely through `sysctl(3)` calls. Likewise, commands such as `ps(1)` and `netstat(1)` rely on `sysctl(2)` to obtain the list of PIDs and active sockets, respectively, though this could be achieved by other means, as well.

kqueues

kqueues are a BSD mechanism for kernel event notifications. A kqueue is a descriptor that blocks until an event of a specific type and category occurs. A user (or kernel) mode process can thus wait on the descriptor, providing a simple but effective method for synchronization of one or more processes.

kqueues and their kevents form the basis for asynchronous I/O in the kernel (and enable the POSIX `poll(2)`/`select(2)`, accordingly). A kqueue can be constructed in user mode by simply calling the `kqueue(2)` system call (#362), with no arguments. Then, the specific events of interest can be specified using the `EV_SET` macro, which initializes a `struct kevent`. Calling the `kevent(2)` or `kevent64(2)` system calls (#363 or #369, respectively) will set the event filters, and return if they have been satisfied. The system supports several "predefined" filters, as shown in Table 3-2:

TABLE 3-2: Some of the predefined Event Filters in <sys/event.h>

EVENT FILTER CONSTANT	USAGE
`EVFILT_MACHPORT`	Monitors a Mach port or port set and returns if a message has been received.
`EVFILT_PROC`	Monitors a specified PID for `execve(2)`, `exit(2)`, `fork(2)`, `wait(2)`, or signals.
`EVFILT_READ`	For files, returns when the file pointer is not at EOF. For sockets, pipes, and FIFOs, returns when there is data to read (such as `select(2)`).

continues

TABLE 3-2 *(continued)*

EVENT FILTER CONSTANT	USAGE
`EVFILT_SESSION`	Monitors an audit session (described in the next section).
`EVFILT_SIGNAL`	Monitors a specific signal to the process, even if the signal is currently ignored by the process.
`EVFILT_TIMER`	A periodic timer with up to nanosecond resolution.
`EVFILT_WRITE`	For files, unsupported. For sockets, pipes, and FIFOs, returns when data may be written. Returns buffer space available in event data.
`EVFILT_VM`	Virtual memory Notifications. Used for memory pressure handling (discussed in Chapter 14).
`EVFILT_VNODE`	Filters file (`vnode`)-specific system calls such as `rename(2)`, `delete(2)`, `unlink(2)`, `link(2)`, and others.

Listing 3-1 demonstrates using `kevents` to track process-level events on a particular PID:

LISTING 3-1: Using kqueues and kevents to filter process events

```
void main (int argc, char **argv)
{
   pid_t pid;  // PID to monitor
   int kq;     // The kqueue file descriptor
   int rc;     // collecting return values
   int done;
   struct kevent ke;

   pid = atoi(argv[1]);

   kq = kqueue();

   if (kq == -1) { perror("kqueue"); exit(2); }

   // Set process fork/exec notifications

   EV_SET(&ke, pid, EVFILT_PROC, EV_ADD,
       NOTE_EXIT | NOTE_FORK | NOTE_EXEC , 0, NULL);

   // Register event

   rc = kevent(kq, &ke, 1, NULL, 0, NULL);
   if (rc < 0) { perror ("kevent"); exit (3); }

   done = 0;
   while (!done) {
```

DX-5562.029

```
            memset(&ke, '\0', sizeof(struct kevent));

            // This blocks until an event matching the filter occurs
            rc = kevent(kq, NULL, 0, &ke, 1, NULL);
            if (rc < 0) { perror ("kevent"); exit (4); }

            if (ke.fflags & NOTE_FORK)
                printf("PID %d fork()ed\n", ke.ident);

            if (ke.fflags & NOTE_EXEC)
                printf("pid %d has exec()ed\n", ke.ident);

            if (ke.fflags & NOTE_EXIT)
               {
                 printf("pid %d has exited\n", ke.ident);
                 done++;
               }

          } // end while
    }
```

Auditing (OS X)

OS X contains an implementation of the Basic Security Module, or BSM. This auditing subsystem originated in Solaris, but has since been ported into numerous UN*X implementations (as *OpenBSM*), among them OS X. This subsystem is useful for tracking user and process actions, though may be costly in terms of disk space and overall performance. It is, therefore, of value in OS X, but less so on a mobile system such as iOS, which is why it is not enabled in the latter.

Auditing, as the security-sensitive operation that it is, must be performed at the kernel level. In BSD and other UN*X flavors the kernel component of auditing communicates with user space via a special character pseudo-device (for example, `/dev/audit`). In OS X, however, auditing is implemented over Mach messages.

The Administrator's View

Auditing is a self-contained subsystem in OS X. The main user-mode component is the `auditd(8)`, a daemon that is started on demand by `launchd(8)`, unless disabled (in the `com.apple.auditd.plist` file). The daemon does not actually write the audit log records; those are done directly by the kernel itself. The daemon does control the kernel component, however, and so he who controls the daemon controls auditing. To do so, the administrator can use the `audit(8)` command, which can initialize (`-i`) or terminate (`-t`) auditing, start a new log (`-n`), or expire (`-e`) old logs. Normally, `auditd(8)` times out after 60 seconds of inactivity (as specified in its plist `TimeOut` key). Just because `auditd(8)` is not running, therefore, implies nothing about the state of auditing.

Audit logs, unless otherwise stated, are collected in `/var/audit`, following a naming convention of `start_time.stop_time`, with the timestamp accurate to the second. Logs are continuously generated, so (aside from crashes and reboots), the `stop_time` of a log is also a `start_time` of its successor. The latest log can be easily spotted by its `stop_time` of `not_terminated`, or a symbolic link to `current`, as shown in Output 3-1.

OUTPUT 3-1: Displaying logs in the /var/audit directory

```
root@Ergo (/)# ls -ld /var/audit
drwx------  3247 root  wheel  110398 Mar 19 17:44 /var/audit


root@Ergo (/)# ls -l /var/audit
…
-r--r-----  1 root  wheel     749 Mar 19 16:33 20120319203254.20120319203327
-r--r-----  1 root  wheel     337 Mar 19 17:44 20120319203327.20120319214427
-r--r-----  1 root  wheel       0 Mar 19 17:44 20120319214427.not_terminated
lrwxr-xr-x  1 root  wheel      40 Mar 19 17:44 current ->
                                                 /var/audit/20120319214427.not_terminated
```

The audit logs are in a compact binary format, which can be deciphered using the `praudit(1)` command. This command can print the records in a variety of human- and machine-readable formats, such as the default CSV or the more elegant XML (using `-x`). To enable searching through audit records, the `auditreduce(1)` command may be used with an array of switches to filter records by event type (`-m`), object access (`-o`), specific UID (`-e`), and more.

Because logs are cycled so frequently, a special character device, `/dev/auditpipe`, exists to allow user-mode programs to access the audit records in real time. The `praudit(1)` command can therefore be used directly on `/dev/auditpipe`, which makes it especially useful for shell scripts. As a quick experiment, try doing so, then locking your screen saver, and authenticating to unlock it. You should see something like Output 3-2.

OUTPUT 3-2: Using praudit(1) on the audit pipe for real-time events

```
root@Ergo (/)# praudit /dev/auditpipe
header,106,11,user authentication,0,Tue Mar 20 02:26:01 2012, + 180 msec
subject,root,morpheus,wheel,root,wheel,38,0,0,0.0.0.0
text,Authentication for user <morpheus>
return,success,0
trailer,106
```

Auditing must be performed at the time of the action, and can therefore have a noticeable impact on system performance as well as disk space. The administrator can therefore tweak auditing using several files, all in `/etc/security`, listed in Table 3-3.

TABLE 3-3: Files in /etc/security Used to Control Audit Policy

AUDIT CONTROL FILE	USED FOR
`audit_class`	Maps event bitmasks to human-readable names, and to the mnemonic classes used in other files for events.
`audit_control`	Specifies audit policy and log housekeeping.

AUDIT CONTROL FILE	USED FOR
`audit_event`	Maps event identifiers to mnemonic class and human-readable name.
`audit_user`	Selectively enables/disables auditing of specific mnemonic event classes on a per-user basis. The record format is: *`Username:classes_audited:classes_not_audited`*
`audit_warn`	A shell script to execute on warnings from the audit daemon (for example, "audit space low (< 5% free) on audit log file-system"). Usually passes the message to `logger(1)`.

The Programmer's View

If auditing is enabled, XNU dedicates system calls #350 through #359 to enable and control auditing, as shown in Table 3-4 (all return the standard `int` return value of a system call: `0` on success, or `-1` and set `errno` on error). On iOS, these calls are merely stubs returning –`ENOSYS` (0x4E).

TABLE 3-4: System Calls Used for Auditing in OS X, BSM-Compliant

#	SYSTEM CALL	USED TO
350	**`audit`**`(const char *rec, u_int length);`	Commit an audit record to the log.
359	**`auditctl`**`(char *path);`	Open a new audit log in file specified by path (similar to `audit -n`)
351	**`auditon`**`(int cmd, void *data, u_int length);`	Configure audit parameters. Accepts various `A_*` commands from `<bsm/audit.h>`.

355
356
	`getaudit` `(auditinfo_t *ainfo);`
	`setaudit` `(auditinfo_t *ainfo);`
		Get or set audit session state. The `auditinfo_t` is defined as
		`struct auditinfo {`
		`au_id_t ai_auid;`
		`au_mask_t ai_mask;`
		`au_tid_t ai_termid;`
		`au_asid_t ai_asid; };`
		These system calls are likely deprecated in Mountain Lion.

continues

TABLE 3-4 *(continued)*

#	SYSTEM CALL	USED TO
357	**`getaudit_addr`** `(auditinfo_addr_t *aa, u_int length);`	As `getaudit` or `setaudit`, but with support for >32-bit `termids`, and an additional 64-bit `ai_flags` field.
358	**`setaudit_addr`** `(auditinfo_addr_t *aa, u_int length);`	
353	**`getauid`**`(au_id_t *auid);`	Get or set the audit session ID.
354	**`setauid`**`(au_id_t *auid);`	

Apple deviates from the BSM standard and enhances it with three additional proprietary system calls, tying the subsystem to the underlying Mach system. Unlike the standard calls, these are undocumented save for their open source implementation, as shown in Table 3-5.

TABLE 3-5: Apple-Specific System Calls Used for Auditing

#	SYSTEM CALL	USED FOR
428	**`mach_port_name_t`** `audit_session_self(void);`	Returns a Mach port (send) for the current audit session
429	**`audit_session_join`** `(mach_port_name_t port);`	Joins the audit session for the given Mach port
432	**`audit_session_port`**`(au_asid_t asid, user_addr_t portnamep);`	New in Lion and relocates `fileport_makeport`. Obtains the Mach port (send) for the given audit session `asid`.

Auditing is revisited from the kernel perspective in Chapter 14.

Mandatory Access Control

FreeBSD 5.x was the first to introduce a powerful security feature known as Mandatory Access Control (MAC). This feature, originally part of Trusted BSD[1], allows for a much more fine-grained security model, which enhances the rather crude UN*X model by adding support for object-level security: limiting access to certain files or resources (sockets, IPC, and so on) by specific processes, not just by permissions. In this way, for example, a specific app could be limited so as not to access the user's private data, or certain websites.

A key concept in MAC is that of a *label*, which corresponds to a predefined classification, which can apply to a set of files or other objects in the system (another way to think of this is as sensitivity tags applied to dossiers in spy movies — "Unclassified," "Confidential," "Top Secret," etc). MAC denies access to any object which does not comply with the label (Sun's swan song, Trusted Solaris, actually made such objects invisible!). OS X extends this further to encompass security policies (for example "No network") that can then be applied to various operations, not just objects.

MAC is a framework — not in the OS X sense, but in the architectural one: it provides a solid foundation into which additional components, which do not necessarily have to be part of the kernel proper, may "plug-in" to control system security. By registering with MAC, specialized kernel extensions can assume responsibility for the enforcement of security policies. From the kernel's side, callouts to MAC are inserted into the various system call implementations, so that each system call must first pass MAC validation, prior to actually servicing the user-mode request. These callouts are only invoked if the kernel is compiled with MAC support, which is on by default in both OS X and iOS. Even then, the callouts return 0 (approving the operation) unless a policy module (specialized kernel extension) has registered for them, and provided its own alternate authorization logic. The MAC layer itself makes no decisions — it calls on the registered policy modules to do so.

The kernel additionally offers dedicated MAC system calls. These are shown in Table 3-6. Most match those of FreeBSD's, while a few are Apple extensions (as noted by the shaded rows).

TABLE 3-6: MAC-Specific System Calls

#	SYSTEM CALL	USED FOR
380	`int __mac_execve(char *fname, char **argp, char **envp, struct mac *mac_p);`	As `execve(2)`, but executes the process under a given MAC label
381	`int __mac_syscall(char *policy, int call, user_addr_t arg);`	MAC-enabled Wrapper for `indirect` syscall.
382 383	`int __mac_[get|set]_file (char *path_p, struct mac *mac_p);`	Get or set label associated with a pathname
384 385	`int __mac_[get|set]_link (char *path_p, struct mac *mac_p);`	Get or set label associated with a link
386 387	`int __mac_[get|set]_proc(struct mac *mac_p);`	Retrieve or set the label of the current process
388 389	`int __mac_[get|set]_fd (int fd, struct mac *mac_p);`	Get or set label associated with a file descriptor. This can be a file, but also a socket or a FIFO
390	`int __mac_get_pid(pid_t pid, struct mac *mac_p);`	Get the label of another process, specified by PID
391	`int __mac_get_lcid(pid_t lcid, struct mac *mac_p);`	Get login context ID
392 393	`int __mac_[get|set]_lctx (struct mac *mac_p);`	Get or set login context ID

continues

TABLE 3-6 *(continued)*

#	SYSTEM CALL	USED FOR
424	`int __mac_mount(char *type, char *path, int flags, caddr_t data, struct mac *mac_p);`	MAC enabled `mount(2)` replacement
425	`int __mac_get_mount(char *path, struct mac *mac_p);`	Get Mount point label information
426	`int __mac_getfsstat(user_addr_t buf, int bufsize, user_addr_t mac, int macsize, int flags);`	MAC enabled `getfsstat(2)` replacement

The administrator can control enforcement of MAC policies on the various subsystems using sysctl(8): MAC dynamically registers and exposes the top-level security MIB, which contain enforcement flags, as shown in Output 3-3:

OUTPUT 3-3: The security sysctl MIBs exposed by MAC, on Lion

```
morpheus@Minion (/)$ sysctl security
security.mac.sandbox.sentinel: .sb-4bde45ee
security.mac.qtn.sandbox_enforce: 1
security.mac.max_slots: 7
security.mac.labelvnodes: 0
security.mac.mmap_revocation: 0         # Revoke mmap access to files on subject relabel
security.mac.mmap_revocation_via_cow: 0 # Revoke mmap access to files via copy on write
security.mac.device_enforce: 1
security.mac.file_enforce: 0
security.mac.iokit_enforce: 0
security.mac.pipe_enforce: 1
security.mac.posixsem_enforce: 1        # Posix semaphores
security.mac.posixshm_enforce: 1        # Posix shared memory
security.mac.proc_enforce: 1            # Process operation (including code signing)
security.mac.socket_enforce: 1
security.mac.system_enforce: 1
security.mac.sysvmsg_enforce: 1
security.mac.sysvsem_enforce: 1
security.mac.sysvshm_enforce: 1
security.mac.vm_enforce: 1
security.mac.vnode_enforce: 1           # VFS VNode operations (including code signing)
```

The proc_enforce and vnode_enforce MIBS are the ones which control, among other things, code signing on iOS. A well known workaround for code signing on jailbroken devices was to manually set both to 0 (i.e. disable their enforcement). Apple made those two settings read only in iOS 4.3 and later, but kernel patching and other methods can still work around this.

DX-5562.035

MAC provides the substrate for OS X's Compartmentalization ("Sandboxing") and iOS's entitlements. Both are unique to OS X and iOS, and are described later in this chapter under "OS X and iOS Security Mechanisms." The kernel perspective of MAC (including an in-depth discussion of its use in OS X and iOS) is described in Chapter 14.

OS X- AND IOS-SPECIFIC TECHNOLOGIES

Mac OS has, over the years, introduced several avant-garde technologies, some of which still remain proprietary. The next section discusses these technologies, particularly the ones that are of interest from an operating-system perspective.

User and Group Management (OS X)

Whereas other UN*X traditionally relies on the age-old password files (`/etc/passwd` and, commonly `/etc/shadow`, used for the password hashes), which are still used in single-user mode (and on iOS), with `/etc/master.passwd` used as the shadow file. In all other cases, however, OS X deprecates them in favor of its own directory service: `DirectoryService(8)` on Snow Leopard, which has been renamed to `opendirectoryd(8)` as of Lion. The daemon's new name reflects its nature: It is an implementation of the OpenLDAP project. Using a standard protocol such as the Lightweight Directory Access Protocol (LDAP) enables integration with non-Apple directory services as well, such as Microsoft's Active Directory. (Despite the "lightweight" moniker, LDAP is a lengthy Internet standard covered by RFCs 4510 through 4519. It is a simplified version of DAP, which is an OSI standard).

The directory service maintains more than just the users and groups: It holds many other aspects of system configuration, as is discussed under "System Configuration" later in the chapter.

To interface with the daemon, OS X supplies a command line utility called `dscl(8)`. You can use this tool, among other things, to display the users and groups on the system. If you try `dscl . -read /Users/username` on yourself (the "." is used to denote the default directory, which is also accessible as `/Local/Default`), you should see something similar to Output 3-4:

OUTPUT 3-4: Running dscl(8) to read user details from the local directory

```
morpheus@ergo(/)$ dscl . -read /Users/ `whoami `
dsAttrTypeNative:_writers_hint: morpheus
dsAttrTypeNative:_writers_jpegphoto: morpheus
dsAttrTypeNative:_writers_LinkedIdentity: morpheus
dsAttrTypeNative:_writers_passwd: morpheus
dsAttrTypeNative:_writers_picture: morpheus
dsAttrTypeNative:_writers_realname: morpheus
dsAttrTypeNative:_writers_UserCertificate: morpheus
AppleMetaNodeLocation: /Local/Default
AuthenticationAuthority: ;ShadowHash; ;Kerberosv5;;morpheus@LKDC:SHA1.3023D12469030DE9DB
FE2C2621A01C121615DC80;LKDC:SHA1.3013D12469030DE9DBFD2C2621A07C123615DC70;
AuthenticationHint:
GeneratedUID: 11E111F7-910C-2410-9BAB-ABB20FE3DF2A
JPEGPhoto:
 ffd8ffe0 00104a46 49460001 01000001 00010000 ffe20238 4943435f 50524f46 494c4500..
```

continues

DX-5562.036

OUTPUT 3-4 *(continued)*

```
 ... User photo in JPEG format
NFSHomeDirectory: /Users/morpheus
Password: ********
PasswordPolicyOptions:
 <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
       <key>failedLoginCount</key>
       <integer>0</integer>
       <key>failedLoginTimestamp</key>
       <date>2001-01-01T00:00:00Z</date>
       <key>lastLoginTimestamp</key>
       <date>2001-01-01T00:00:00Z</date>
       <key>passwordTimestamp</key>
       <date>2011-09-24T20:23:03Z</date>
</dict>
</plist>
Picture:
 /Library/User Pictures/Fun/Smack.tif
PrimaryGroupID: 20
RealName: Me
RecordName: morpheus
RecordType: dsRecTypeStandard:Users
UniqueID: 501
UserShell: /bin/zsh
```

You can also use the `dscl(8)` tool to update the directory and create new users. The shell script in Listing 3-2 demonstrates the implementation of a command-line `adduser`, which OS X does not provide.

LISTING 3-2: A script to perform the function of adduser (to be run as root)

```
#!/bin/bash
# Get username, ID and full name field as arguments from command line
USER=$1
ID=$2
FULLNAME=$3
# Create the user node
dscl . -create /Users/$USER
# Set default shell to zsh
dscl . -create /Users/$USER UserShell /bin/zsh
# Set GECOS (full name for finger)
dscl . -create /Users/$USER RealName "$FULLNAME"
dscl . -create /Users/$USER UniqueID $ID
# Assign user to gid of localaccounts
dscl . -create /Users/$USER PrimaryGroupID 61
# Set home dir (~$USER)
dscl . -create /Users/$USER NFSHomeDirectory /Users/$USER
```

```
# Make sure home directory is valid, and owned by the user
mkdir /Users/$USER
chown $USER /Users/$USER
# Optional: Set the password.
dscl . -passwd /Users/$USER "changeme"
# Optional: Add to admin group
dscl . -append /Groups/admin GroupMembership $USER
```

One of Lion's early security vulnerabilities was that `dscl(8)` *could be used to change passwords of users without knowing their existing passwords, even as a non-root user. If you keep your OS X constantly updated, chances are this issue has been resolved by a security update.*

The standard UNIX utilities of `chfn(1)` *and* `chsh(1)`*, which enable the modification of the full name and shell for a given user, respectively, are implemented transparently over directory services by launching the default editor to allow root to type in the fields, rather than bother with* `dscl(8)` *directly. Most administrators, of course, probably use the system configuration GUI — a much safer option, though not as scalable when one needs to create more than a few users.*

System Configuration

Much like it deprecates `/etc` user database files, OS X does away with most other configuration files, which are traditionally used in UN*X as the system "registry."

To maintain system configuration, OS X and iOS use a specialized daemon: – `configd(8)`. This daemon can load additional loadable bundles ("plug-ins") located in the `/System/Library/SystemConfiguration/` directory, which include IP and IPv6 configuration, logging, and other bundles. The average user, of course, is blissfully unaware of this, as the System Preferences application can be used as a graphical front-end to all the configuration tasks.

Command line-oriented power users can employ a specialized tool, `scutil(8)` in order to navigate and query the system configuration. This interactive utility can list and show keys as shown in the following code snippet:

```
root@Padishah (~)# scutil
> list
  subKey [0] = Plugin:IPConfiguration
  subKey [1] = Plugin:InterfaceNamer
  subKey [2] = Setup:
  subKey [3] = Setup:/
  subKey [4] = Setup:/Network/Global/IPv4
  subKey [5] = Setup:/Network/HostNames
  ...
  subKey [50] = com.apple.MobileBluetooth
  subKey [51] = com.apple.MobileInternetSharing
  subKey [52] = com.apple.network.identification

> show com.apple.network.identification
<dictionary> {
  ActiveIdentifiers : <array> {
    0 : IPv4.Router=192.168.1.254;IPv4.RouterHardwareAddress=00:43:a3:f2:81:d9
  }
```

DX-5562.038

```
  PrimaryIPv4Identifier : IPv4.Router=192.168.1.254;IPv4.RouterHardwareAddress=
00:43:a3:f2:81:d9
  ServiceIdentifiers : <array> {
    0 : 12C4C9CC-7E42-1D2D-ACF6-AAF7FFAF2BFC
  }
}
```

The public `SystemConfiguration.framework` allows programmatic access to the system configuration. Commands such as OS X's `pmset(1)`, which configures power management settings, link with this framework. The framework exists in OS X and iOS, so the program shown in Listing 3-3 can compile and run on both.

LISTING 3-3: Using the SystemConfiguration APIs to query values

```
#include <SystemConfiguration/SCPreferences.h>
// Also implicitly uses CoreFoundation/CoreFoundation.h

void dumpDict(CFDictionaryRef dict){
    // Quick and dirty way of dumping a dictionary as XML
    CFDataRef xml = CFPropertyListCreateXMLData(kCFAllocatorDefault,
                                                (CFPropertyListRef)dict);
    if (xml) {
        write(1, CFDataGetBytePtr(xml), CFDataGetLength(xml));
        CFRelease(xml);
    }
}

void main (int argc, char **argv)
{
  CFStringRef myName = CFSTR("com.technologeeks.SystemConfigurationTest");
  CFArrayRef  keyList;
  SCPreferencesRef prefs = NULL;
  char *val;
  CFIndex i;
  CFDictionaryRef global;

  // Open a preferences session
  prefs = SCPreferencesCreate (NULL,   // CFAllocatorRef allocator,
                               myName, // CFStringRef name,
                               NULL);  // CFStringRef prefsID

  if (!prefs) { fprintf (stderr,"SCPreferencesCreate"); exit(1); }

  // retrieve preference namespaces
  keyList =  SCPreferencesCopyKeyList (prefs);

  if (!keyList) { fprintf (stderr,"CopyKeyList failed\n"); exit(2);}

  // dump 'em
  for (i = 0; i < CFArrayGetCount(keyList); i++) {
       dumpDict(SCPreferencesGetValue(prefs, CFArrayGetValueAtIndex(keyList, i)));
       }

}
```

The dictionaries dumped by this program are naturally maintained in plist files. The default location for these dictionaries is in `/Library/Preferences/SystemConfiguration`. If you compare the output of this program with that of the `preferences.plist` file from that directory, you will see it matches.

Experiment: Using scutil(8) for Network Notifications

You can also use the `scutil(8)` command to watch for system configuration changes, as demonstrated in the following experiment:

1. **Using `scutil(8)`, set a watch on the state of the Airport interface** (if you have one, otherwise the primary Ethernet interface will do):

```
> n.add State:/Network/Interface/en0/AirPort
> n.watch
#  verify the notification was added
> n.list
  notifier key [0] = State:/Network/Interface/en0/AirPort
```

2. **Disable Airport (or unplug your network cable).** You should see notification messages break through the `scutil` prompt:

```
notification callback (store address = 0x10010a150).
  changed key [0] = State:/Network/Interface/en0/AirPort
notification callback (store address = 0x10010a150).
  changed key [0] = State:/Network/Interface/en0/AirPort
notification callback (store address = 0x10010a150).
  changed key [0] = State:/Network/Interface/en0/AirPort
```

3. **Use the "show" subcommand to see the changed key.** In this case, the power status value has been changed:

```
> show State:/Network/Interface/en0/AirPort
<dictionary> {
  Power Status : 0
  SecureIBSSEnabled : FALSE
  BSSID : <data> 0x0013d37f84d9
  Busy : FALSE
  SSID_STR : AAAA
  SSID : <data> 0x41414141
  CHANNEL : <dictionary> {
    CHANNEL : 11
    CHANNEL_FLAGS : 10
  }
}
```

In order to watch for changes programmatically, you can use the `SCDynamicStore` class. Because obtaining the network connectivity status is a common action, Apple provides the far simpler `SCNetworkReachability` class. Apple Developer also provides sample code demonstrating the usage of the class.[2]

Logging

With the move to a BSD-based platform, OS X also inherited support for the traditional UNIX System log. This support (detailed in Apple Technical Article TA26117[3]) provides the full compatibility with the ages-old mechanism commonly referred to as `syslogd(8)`.

DX-5562.040

The `syslog` mechanism is well detailed in many other references (including the aforementioned technical article). In a nutshell, it handles textual messages, which are classified by a message *facility* and *severity*. The facility is the class of the reporting element: essentially, the message source. The various UNIX subsystems (mail, printing, cron, and so on) all have their own facilities, as does the kernel (`LOG_KERN`, or "kern"). Severities range from `LOG_DEBUG` and `LOG_INFO` ("About to open file…"), through `LOG_ERR` ("Unable to open file"), `LOG_CRIT` ("Is that a bad sector?"), `LOG_ALERT` ("Hey, where's the disk?!"), and finally, to `LOG_EMERG` ("Meltdown imminent!"). By using the configuration file `/etc/syslog.conf`, the administrator can decide on *actions* to take, corresponding to facility/severity combinations. Actions include the following:

- Message certain usernames specified
- Log to files or devices (specified as a full path, starting with "`/`" so as to disambiguate files from usernames)
- Pipe to commands (`|/path/to/program`)
- Send to a network host (`@`*`loghost`*)

Programmers interface with `syslog` using the `syslog(3)` API, consisting of a call to `openlog()` (specifying their name, facility, and other options), through `syslog()`, which logs the messages with a given priority. The `syslog` daemon intercepts the messages through a UNIX domain socket (traditionally `/dev/log`, though in OS X this has been changed to `/var/run/syslog`).

OS X 10.4 (Tiger) introduced a new model for logging called the Apple System Log, or ASL. This new architecture (which is also used in iOS) aims to provide more flexibility than is provided by `syslog`. ASL is modeled after `syslog`, with the same levels and severities, but allows more features, such as filtering and searching not offered by `syslog`.

ASL is modular in that it simultaneously offers four logging interfaces:

- **The backward-compatible syslogd:** Referred to as BSD logging, ASL can be configured to accept `syslog` messages (using `–bsd_in 1`), and process them according to `/etc/syslog.conf` (using `–bsd_out 1`). In OS X, these are enabled by default, but *not so on iOS*. The messages, as in `syslogd`, come in through the `/var/run/syslog` socket.
- **The network protocol syslogd:** On the well-known UDP port 514, this protocol may be enabled by `–udp_in 1`. It is actually enabled by default, but ASL/`syslogd` relies on `launchd(8)` for its socket handling, and therefore the socket is not active by default.
- **The kernel logging interface:** Enabled (the default) by `–klog_in 1`, this interface accepts kernel messages from `/dev/log` (a character device, incorrectly specified in the documentation as a UNIX domain socket).
- **The new ASL interface:** By using `–asl_in 1`, which is naturally enabled by default, ASL messages can be obtained from clients of the `asl(3)` API using `asl_log(3)` and friends. These messages come in through the `/var/run/asl_input` socket, and are of a different format than the `syslogd` ones (hence the need for two separate sockets).

ASL logs are collected in `/var/log/asl`. They are managed (rotated/deleted) by the `aslmanager(8)` command, which is automatically run by `launchd` (from `com.apple.aslmanager.plist`). You may also run the command manually.

ASL logs, unlike syslog files, are binary, not text. This makes them somewhat smaller in size, but not as `grep(1)`-friendly as `syslog`'s. Apple includes the `syslog(1)` command in OS X to display and view logs, as well as perform searches and filters.

Experiment: Enabling System Logging on a Jailbroken iOS

Apple has intentionally disabled the legacy BSD `syslog` interface, but re-enabling it is a fairly simple matter for the root user via a few simple steps:

1. **Create an `/etc/syslog.conf` file.** The easiest way to create a valid file is to simply copy a file from an OS X installation. The default `syslog.conf` looks something like Listing 3-4:

LISTING 3-4: A default /etc/syslog.conf, from an OS X system

```
*.notice;authpriv,remoteauth,ftp,install,internal.none      /var/log/system.log
kern.*                                                      /var/log/kernel.log

# Send messages normally sent to the console also to the serial port.
# To stop messages from being sent out the serial port, comment out this line.
#*.err;kern.*;auth.notice;authpriv,remoteauth.none;mail.crit        /dev/tty.serial

# The authpriv log file should be restricted access; these
# messages shouldn't go to terminals or publically-readable
# files.
auth.info;authpriv.*;remoteauth.crit                       /var/log/secure.log

lpr.info                                                   /var/log/lpr.log
mail.*                                                     /var/log/mail.log
ftp.*                                                      /var/log/ftp.log
install.*                                                  /var/log/install.log
install.*                                                  @127.0.0.1:32376
local0.*                                                   /var/log/appfirewall.log
local1.*                                                   /var/log/ipfw.log

*.emerg                                                     *
```

2. **Enable the `-bsd_out` switch for syslogd.** The `syslogd` process is started both in iOS and OS X by `launchd(8)`. To change its startup parameters, you must modify its property list file. This file is aptly named `com.apple.syslogd.plist`, and you can find it in the standard location for all launch daemons: `/System/Library/LaunchDaemons`.

   The file, however, like all plists on iOS, is in binary form. Copy the file to `/tmp` and use `plutil -convert xml1` to change it to the more readable XML form. After it is in XML, just edit it so that the `ProgramArguments` key contains `–bsd_out 1`. Because the key expects an array, the arguments have to be written separately, as follows:

```
<key>ProgramArguments</key>
        <array>
               <string>/usr/sbin/syslogd</string>
               <string>-bsd_out</string>
               <string>1</string>
        </array>
```

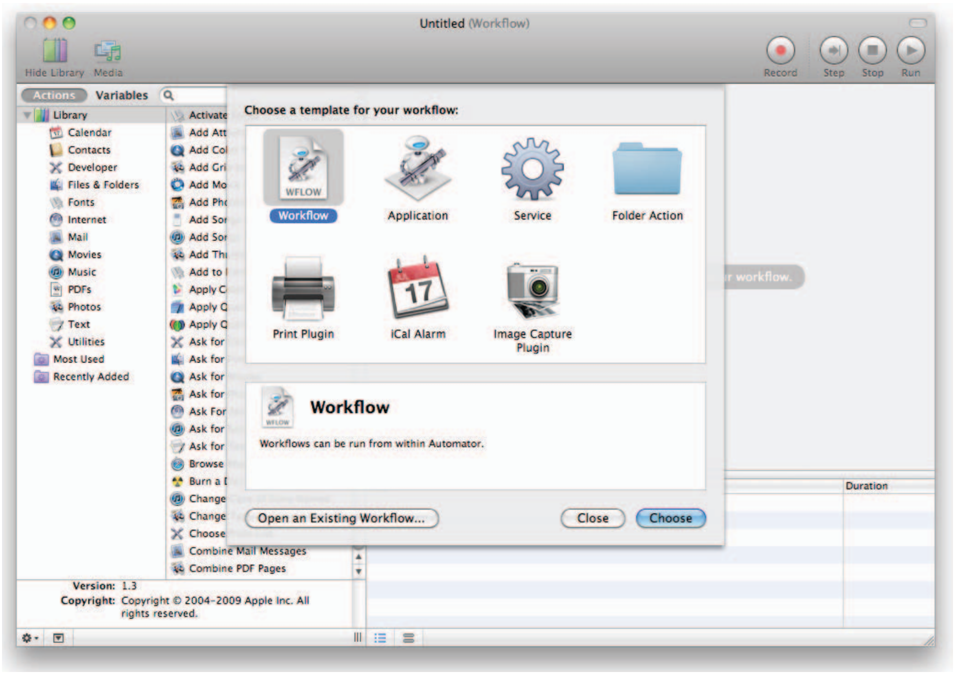
   After this is done, convert the file back to the binary format (`plutil -convert binary1` should do the trick), and copy it back to `/System/Library/LaunchDaemons`.

3. **Restart `launchd`, and then `syslogd`.** A `kill -HUP 1` will take care of `launchd`, and — after you find the process ID of `syslogd` — a kill `-TERM` on its PID will cause `launchd` to restart it, this time with the `-bsd_out 1` argument, as desired. A `ps aux` will verify that is indeed the case, as will the log files in `/var/log`.

Apple Events and AppleScript

One of OS X's oft-overlooked, though truly powerful features, lies in its scripting capabilities. AppleScript has its origins traced back to OS 7(!) and a language called HyperCard. It has since evolved considerably, and become the all-powerful mechanism behind the `osascript(1)` command and the friendly (but neglected) Automator.

In a somewhat similar way to how iPhone's SIRI recognizes English patterns, AppleScript allows a semi-natural language interface to scriptable applications. The "semi" is because commands must follow a given grammar. If the grammar is adhered to, however, it allows for a large range of freedom. The OS X built-in applications can be almost fully automated. For those wary of scripts, the Automator provides a feature-oriented drag-and-drop GUI, as shown in Figure 3-1. Note the rich "Library" composed of actions and definitions in `/System/Library/Automator`.



Choose a template for your workflow:

Workflow

Workflows can be run from within Automator.

Version: 1.3
Copyright: Copyright © 2004-2009 Apple Inc. All rights reserved.

FIGURE 3-1: Automator and its built-in templates.

DX-5562.043

The mechanism allowing AppleScript's magic is called AppleEvents. AppleScript can be extended to remote hosts, either via the (now obsolete) AppleTalk protocol, or over TCP/IP. In the latter case, the protocol is known as "eppc," and is a proprietary, undocumented protocol that uses TCP port 3031. The remote functionality is only enabled if Remote Apple Events are enabled from the Sharing applet of System Preferences. This tells `launchd(8)` to listen on the eppc port, and — when requests are received — start the AppleEvents server, `AEServer` (found in the `Support/` directory of the `AE.framework`, which is internal to CoreServices). `launchd(8)` is responsible for starting many on-demand services from their respective plist files in `/System/Library/LaunchDaemons`. `AEServer`'s is `com.apple.eppc.plist`.

Though covering it is far beyond the scope of this book, AppleScript is a great mechanism for automating tasks. Outside Apple's own reference, two books devoted to the topic can be found elsewhere.[4,5] The simple experiment described next, however, shows you the flurry of events that occurs behind the scenes when you run AppleScript or Automator.

Experiment: Viewing Apple Events

You can easily see what goes on in the Apple Events plane via two simple environment variables — `AEDebugSends` and `AEDebugReceives`. Then, using `osascript` (or, in some cases, Automator), will generate plenty of output. In Output 3-5, note the debug info only pertains to events sent or received by the shell and its children, not events occurring elsewhere in the system.

OUTPUT 3-5: Output of AppleEvents driving Safari application launch

```
morpheus@ergo(/)$ export AEDebugSends=1 AEDebugReceives=1
morpheus@ergo(/)$ osascript -e 'tell app "Safari" to activate'
{ 1 } 'aevt':  ascr/gdte (i386){
          return id: -16316 (0xffffc044)
     transaction id: 0 (0x0)
  interaction level: 64 (0x40)
     reply required: 1 (0x1)
             remote: 0 (0x0)
      for recording: 0 (0x0)
         reply port: 0 (0x0)
  target:
    { 2 } 'psn ':  8 bytes {
      { 0x0, 0x5af5af } (Safari)
    }
  fEventSourcePSN: { 0x1,0xc044 } ()
  optional attributes:
    < empty record >
  event data:
    { 1 } 'aevt':  - 1 items {
      key '----' -
        { 1 } 'long':  4 bytes {
          0 (0x0)
        }
    }
}
```

continues

OUTPUT 3-5 *(continued)*

```
{ 1 } 'aevt':  aevt/ansr (****){
          return id: -16316 (0xffffc044)
     transaction id: 0 (0x0)
  interaction level: 112 (0x70)
     reply required: 0 (0x0)
             remote: 0 (0x0)
      for recording: 0 (0x0)
         reply port: 0 (0x0)
  target:
    { 1 } 'psn ':  8 bytes {
      { 0x1, 0xc044 } (<process { 1, 49220 } not found>
      )
    }
  fEventSourcePSN: { 0x0,0x5af5af } (Safari)
  optional attributes:
    < empty record >
  event data:
    { 1 } 'aevt':  - 1 items {
      key '----' -
        { 1 } 'aete':  9952 bytes {
           000: 0100 0000  0000 0500  0a54 7970  6520 4e61     ........-Type Na
           001: 6d65 731a  4f74 6865  7220 636c  6173 7365     mes.Other classe
           ...:  // etc, etc, etc…
```

FSEvents

All modern operating systems offer their developers APIs for file system notification. These enable quick and easy response by user programs for additions, modifications, and deletions of files. Thus, Windows has its `MJ_DIRECTORY_CONTROL`, Linux has `inotify`. Mac OS X and iOS (as of version 5.0) both offer FSEvents.

FSEvents is conceptually somewhat similar to Linux's `inotify` — in both, a process (or thread) obtains a file descriptor, and attempts to `read(2)` from it. The system call blocks until some event occurs — at which time the received buffer contains the event details by which the program can tell what happened, and then act accordingly (for example, display a new icon in the file browser).

FSEvents is, however, a tad more complicated (and, some would say, more elegant) than inotify. In it, the process proceeds as follows:

- The process (or thread) requests to get a handle to the FSEvents mechanism. This is `/dev/fsevents`, a pseudo-device.
- The requestor then issues a special `ioctl(2)`, `FSEVENTS_CLONE`. This `ioctl` enables the specific filtering of events so that only events of interest — specific operations on particular files — are delivered. Table 3-7 lists the types that are currently supported. Supporting these events is possible because FSEvents is plugged into the kernel's file system-handling logic (VFS, the Virtual File system Switch — see Chapter 15 for more on that topic). Each and every supported event will add a pending notification to the cloned file descriptor.

DX-5562.045

TABLE 3-7: FSEvent Types

FSEVENT CONSTANT	INDICATES
`FSE_CREATE_FILE`	File creation.
`FSE_DELETE`	File/directory has been removed.
`FSE_STAT_CHANGED`	`stat(2)` of file or directory has been changed.
`FSE_RENAME`	File/directory has been renamed.
`FSE_CONTENT_MODIFIED`	File has been modified.
`FSE_EXCHANGE`	The `exchangedata(2)` system call.
`FSE_FINDER_INFO_CHANGED`	File finder information attributes have changed.
`FSE_CREATE_DIR`	A new directory has been created.
`FSE_CHOWN`	File/directory ownership change.
`FSE_XATTR_MODIFIED`	File/directory extended attributes have been modified.
`FSE_XATTR_REMOVED`	File/directory extended attributes have been removed.

- Using `ioctl(2)`, the watcher can modify the exact event details requested in the notification. The control codes defined include `FSEVENTS_WANT_COMPACT_EVENTS` (to get less information), `FSEVENTS_WANT_EXTENDED_INFO` (to get even more information), and `NEW_FSEVENTS_DEVICE_FILTER` (to filter on devices the watcher is not interested in watching).
- The requestor (also called the "watcher") then enters a `read(2)` loop. Each time the system call returns, it populates the user-provided buffer with an array of event records. The read can be tricky, because a single operation might return multiple records of variable size. If events have been dropped (due to kernel buffers being exceeded), a special event (`FSE_EVENTS_DROPPED`) will be added to the event records.

If you check Apple's documentation, the manual pages, or the include files, your search will come out quite empty handed. `<sys/fsevents.h>` did make an early cameo appearance when FSEvents was introduced, but has since been thinned and deprecated (and might disappear in Mountain Lion altogether). This is because, even though the API remains public, it only has some three official users:

- `coreservicesd`: This is an Apple internal daemon supporting aspects of Core Services, such as launch services and others.
- `mds`: The Spotlight server. Spotlight is a "heavy" user of FSEvents, relying on notifications to find and index new files.
- `fseventsd`: A generic user space daemon that is buried inside the CoreServices framework (alongside `coreservicesd`). FSEventsd can be told to not log events by a "`no_log`" file in the `.fseventsd` directory, which is created on the root of every volume.

Both Objective-C and C applications can use the CoreServices Framework (Carbon) APIs of `FSEventStreamCreate` and friends. This framework is a thin layer on top of the actual mechanism,

DX-5562.046

which allows integration of the "real" API with the RunLoop model, events, and callbacks. In essence, this involves converting the blocking, synchronous model to an asynchronous, event-driven one. Apple documents this well.[6] The rest of this section, therefore, concentrates on the lower-level APIs.

Experiment: A File System Event Monitor

Listing 3-5 shows a barebones FSEvents client that will listen on a particular path (given as an argument) and display events occurring on the path. Though functionally similar to `fs_usage(1)`, the latter does not use FSEvents (it uses the little-documented `kdebug` API, described in Chapter 5, "Process Tracing and Debugging").

LISTING 3-5: A bare bones FSEvents-based file monitor

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/ioctl.h>     // for _IOW, a macro required by FSEVENTS_CLONE
#include <sys/types.h>     // for uint32_t and friends, on which fsevents.h relies
#include <sys/fsevents.h>

// The struct definitions are taken from bsd/vfs/vfs_events.c
// since they are no long public in <sys/fsevents.h>

#pragma pack(1)
typedef struct kfs_event_a {
  uint16_t type;
  uint16_t refcount;
  pid_t    pid;
} kfs_event_a;

typedef struct kfs_event_arg {
  uint16_t type;
  uint16_t pathlen;
  char data[0];
} kfs_event_arg;

#pragma pack()

int print_event (void *buf, int off)
{
   // Simple function to print event - currently a simple printf of "event!".
   // The reader is encouraged to improve this, as an exercise.
   // This book's website has a much better (and longer) implementation
   printf("Event!\n");
   return (off);

}

void main (int argc, char **argv)
{

        int fsed, cloned_fsed;
        int i;
```

DX-5562.047

```
int rc;
fsevent_clone_args  clone_args;
char buf[BUFSIZE];

fsed = open ("/dev/fsevents", O_RDONLY);

int8_t  events[FSE_MAX_EVENTS];


if (fsed < 0)
{
        perror ("open"); exit(1);
}


// Prepare event mask list. In our simple example, we want everything
// (i.e. all events, so we say "FSE_REPORT" all). Otherwise, we
// would have to specifically toggle FSE_IGNORE for each:
//
// e.g.
//       events[FSE_XATTR_MODIFIED] = FSE_IGNORE;
//       events[FSE_XATTR_REMOVED]  = FSE_IGNORE;
// etc..

for (i = 0; i < FSE_MAX_EVENTS; i++)
{
       events[i] = FSE_REPORT;
}

memset(&clone_args, '\0', sizeof(clone_args));
clone_args.fd = &cloned_fsed; // This is the descriptor we get back
clone_args.event_queue_depth = 10;
clone_args.event_list = events;
clone_args.num_events = FSE_MAX_EVENTS;

// Request our own fsevents handle, cloned

rc = ioctl (fsed, FSEVENTS_CLONE, &clone_args);

if (rc < 0) { perror ("ioctl"); exit(2);}
printf ("So far, so good!\n");
close (fsed);

while ((rc = read (cloned_fsed, buf, BUFSIZE)) > 0)
{
        // rc returns the count of bytes for one or more events:
        int offInBuf = 0;

        while (offInBuf < rc) {

           struct kfs_event_a *fse = (struct kfs_event_a *)(buf + offInBuf);
           struct kfs_event_arg *fse_arg;

           struct fse_info *fse_inf;

        if (offInBuf) { printf ("Next event: %d\n", offInBuf);};
```

continues

LISTING 3-5 *(continued)*

```
                offInBuf += print_event(buf,offInBuf); // defined elsewhere


                } // end while offInBuf..
                if (rc != offInBuf)
                   { fprintf (stderr, "***Warning: Some events may be lost\n"); }


        } // end while rc = ..

} // end main
```

If you compile this example on either OS X or iOS 5 and, in another terminal, make some file modifications (for example, by creating a temporary file), you should see printouts of file system event occurrences. In fact, even if you don't do anything, the system periodically creates and deletes files, and you will be able to receive notifications.

Note this fairly rudimentary example can be improved on in many ways, not the least of which is display event details. Singh's book has an "fslogger" application (which no longer compiles on Snow Leopard due to missing dependencies). One nifty GUI-based app is FernLightning's "fseventer," [7] which is conceptually very similar to this example, but whose interface is far richer (yet has not been updated in recent years). The book's companion website offers a tool, filemon, which improves this example and can prove quite useful, especially on iOS 5. Output 3-6 shows a sample output of this tool.

OUTPUT 3-6: Output of an fsevents-based file monitoring tool

```
File /private/tmp/xxxxx has been modified
     PID: 174 (/tmp/a)
     INODE: 7219206 DEV 40007 UID 501 (morpheus) GID 501
File /Users/morpheus/Library/PubSub/Database/Database.sqlite3-journal has been created
     PID: 43397 (mysqld)
     INODE: 7219232 DEV 40007 UID 501 (morpheus) GID 501
File /Users/morpheus/Library/PubSub/Database/Database.sqlite3-journal has been modified
     PID: 43397 (mysqld)
     INODE: 7219232 DEV 40007 UID 501 (morpheus) GID 501
File /Users/morpheus/Library/PubSub/Database/Database.sqlite3-journal has been deleted
Type: 1 (Deleted ) refcount 0  PID: 43397
     PID: 43397 (mysqld)
     INODE: 7219232 DEV 40007 UID 501 (morpheus) GID 501
...
```

Notifications

OS X provides a systemwide notification mechanism. This is a form of distributed IPC, by means of which processes can broadcast or listen on events. The heart of this mechanism is the `notifyd(8)` daemon, which is started at boot time: this is the Darwin notification server. An additional daemon, `distnoted(8)`, functions as the distributed notification server. Applications may use the `notify(3)` API to pass messages to and from the daemons. The messages are for given names, and Apple recommends the use of reverse DNS namespaces here, as well (for example, *com.myCompany.myNotification*) to avoid any collisions.

DX-5562.049

The API is very versatile and allows requesting notifications by one of several methods. The well-documented `<notify.h>` lists functions to enable the notifications over UNIX signals, Mach ports, and file descriptors. Clients may also manually suspend or resume notifications. The `notifyd(8)` handles most notifications, by default using Mach messages and registering the Mach port of `com.apple.system.notification_center`.

A command line utility, `notifyutil(1)`, is available for debugging. Using this utility, you can wait for (`-w`) and post (`-p`) notifications on arbitrary keys.

An interesting feature of `notifyd(8)` is that it is one of the scant few daemons to use Apple's file-port API. This enables file descriptors to be passed over Mach messages.

Additional APIs of interest

Additional Apple-specific APIs worth noting, but described elsewhere in this book include:

- **Grand Central Dispatch (Chapter 4):** A system framework for parallelization using work queue extensions built on top of pthread APIs.
- **The Launch Daemon (Chapter 7):** Fusing together many of UN*X system daemons (such as init, inetd, at, crond and others), along with the Mach bootstrap server.
- **XPC (Chapter 7):** A framework for advanced IPC, enabling privilege separation between processes
- **kdebug (Chapter 5):** A little-known yet largely-useful facility for kernel-level tracing of system calls and Mach traps.
- **System sockets (Chapter 17):** Sockets in the `PF_SYSTEM` namespace, which allow communication with kernel mode components
- **Mach APIs (Chapters 9, 10, and 11):** Direct interfaces to the Mach core of XNU, which supply functionality matching the higher level BSD/POSIX interfaces, but in some cases well exceeding them.
- **The IOKit APIs (Chapter 19):** APIs to communicate with device drivers, providing a plethora of diagnostics information as well as powerful capabilities for controlling drivers from user mode.

OS X AND IOS SECURITY MECHANISMS

Viruses and malware are rare on OS X, which is something Apple has kept boasting for many years as an advantage for Mac, in their commercials of "Mac versus PC." This, however, is largely due to the Windows monoculture. Put yourself in the role of Malware developer, concocting your scheme for the next devious bot. Would you invest time and effort in attacking over 90% of the world, or under 5%?

Indeed, OS X (and, to an extent, Linux) remain healthy, in part, simply because they do not attract much attention from malware "providers" (another reason is that UN*X has always adhered to the principle of least privilege, in this case not allowing the user root access by default). This, however, is changing, as with OS X's slow but steady increase in market share, so increases its allure for malware. The latest Mac virus, "Flashback" (so called because it is a Trojan masquerading as an Adobe Flash update) infected some 600,000 users in the United States alone. Certain industry experts were quick to pillory Apple for its hubris, chiding their security mechanisms as being woefully inefficient and backdated.

DX-5562.050

In actuality, however, Apple's application security is light years (if not parsecs) ahead of its peers. Windows' User Account Control (UAC) has been long present in OS X. iOS's hardening makes Android seem riddled in comparison. Nearly all so called "viruses" which do exist in Mac are actually Trojans — which rely on the cooperation (and often utter gullibility) of the unwitting user. Apple is well aware of that, and is determined to combat malware. The arsenal with which to do that has been around since Leopard, and Apple is investing ongoing efforts to upgrade it in OS X and, even more so in iOS.

Code Signing

Before software can be secured, its origin must be *authenticated*. If an app is downloaded from some random site on the Internet, there is a significant risk it is actually malware. The risk is greatly mitigated, however, if the software's origin can be verifiably determined, and it can further be assured that it has not been modified in transit.

Code signing provides the mechanism to do just that. Using the same X.509v3 certificates that SSL uses to establish the identity of websites (by signing their public key with the private key of the issuer), Apple encourages developers to sign their applications and authenticate their identity. Since the crux of a digital signature is that the signer's public key must be a priori known to the verifier, Apple embeds its certificates into both OS X and iOS's keychains (much like Microsoft does in Windows), and is effectively the only root authority. You can easily verify this using the `security(1)` utility, which (among its many other functions) can dump the system keychains, as shown in Output 3-7:

OUTPUT 3-7: Using security(1) to display Apple's built-in certificates on OS X

```
morpheus@Minion (~)$ security -i    # Interactive mode
security> list-keychains
  "/Users/morpheus/Library/Keychains/login.keychain" # User's passwords, etc
  "/Library/Keychains/System.keychain"               # Wi-Fi password,s and certificates

                                    # Non-Interactive mode

morpheus@Minion (~)$ security dump-keychain /Library/Keychains/System.keychain |
                      grep labl                       # Show only labels
    "labl"<blob>="com.apple.systemdefault"
    "labl"<blob>="com.apple.kerberos.kdc"
    "labl"<blob>="Apple Code Signing Certification Authority"
    "labl"<blob>="Software Signing"
    "labl"<blob>="Apple Worldwide Developer Relations Certification Authority"
```

Apple has developed a special language to define code signing requirements, which may be displayed with the `csreq(1)` command. Apple also provides the `codesign(1)` command to allow developers to sign their apps (as well as verify/display existing signatures), but `codesign(1)` won't sign anything without a valid, trusted certificate, which developers can only obtain by registering with Apple's Developer Program. Apple's Code Signing Guide[8] covers the code signing process in depth, with Technical Note 2250[9] discussing iOS.

Whereas in OS X code signing is optional, in iOS it is very much mandatory. If, by some miracle, an unsigned application makes its way to the file system, it will be killed by the kernel upon any attempted execution. This is what makes jailbreakers' life so hard: The system simply refuses to run

DX-5562.051

unsigned code, and so the only way in is by exploiting vulnerabilities in existing, signed applications (and later the kernel itself). Jailbreakers must therefore seek faults in iOS's system apps and libraries (e.g. MobileSafari, Racoon, and others). Alternatively, they may seek faults in the code-signing mechanism itself, as was done by renowned security researcher Charlie Miller in iOS 5.0.[10] Disclosing this to Apple, however, proved a Pyrrhic victory. Apple quickly patched the vulnerability in 5.0.1, and another future jailbreak door slammed shut forever. Mr. Miller himself was controversially banned from the iOS Developer Program.

Code-signed applications may still be malicious. Any applications that violate the terms of service, however, would quickly lead to their developer becoming a persona non grata at Apple, banned from the Mac/iOS App Stores (q.v. Mr. Miller). Since registering with Apple involves disclosing personal details, these malicious developers could also be the target of a lawsuit. This is why you won't find any apps in iOS's App Store attempting to spawn `/bin/bash` or mimic its functionality. Nobody wants to get on Apple's bad side.

Compartmentalization (Sandboxing)

Originally considered a vanguard, nice-to-have feature, *compartmentalization* is becoming an integral part of the Apple landscape. The idea is a simple, yet principal tenet of application security: Untrusted applications must run in a compartment, effectively a quarantined environment wherein all operations are subject to restriction. Formerly known in Leopard as *seatbelt*, the mechanism has since been renamed *sandbox*, and has been greatly improved in Lion, touted as one of its stronger suits. A thorough discussion of the sandbox mechanism (as it was implemented in Snow Leopard) can be found in Dionysus Blazakis's Black Hat DC 2011 presentation[11], though the sandbox has undergone significant improvements since.

iOS — the Sandbox as a jail

In iOS, the sandbox has been integrated tightly since inception, and has been enhanced further to create the "jail" which the "jailbreakers" struggle so hard to break. The limitations in an App's "jail" include, but are not limited to:

- Inability to break out of the app's directory. The app effectively sees its own directory (`/var/mobile/Applications/<app-GUID>`) as the root, similar to the `chroot(2)` system call. As a corollary, the app has no knowledge of any other installed apps, and cannot access system files.
- Inability to access any other process on the system, even if that process is owned by the same UID. The app effectively sees itself as the only process executing on the system.
- Inability to directly use any of the hardware devices (camera, GPS, and others) without going through Apple's Frameworks (which, in turn, can impose limitations, such as the familiar user prompts).
- Inability to dynamically generate code. The low-level implementations of the `mmap(2)` and `mprotect(2)` system calls (Mach's `vm_map_enter` and `vm_map_protect`, respectively, as discussed in Chapter 13) are intentionally modified to circumvent any attempts to make writable memory pages also executable. This is discussed in Chapter 11.
- Inability to perform any operations but a subset of the ones allowed for the user mobile. Root permissions for an app (aside for Apple's own) are unheard of.

Entitlements (discussed later) can release some well-behaving apps from solitary confinement, and some of Apple's own applications do possess root privileges.

Voluntary Imprisonment

Execution in a sandbox is still voluntary (at least, in OS X). A process must willingly call `sandbox_init(3)` to enter a sandbox, with one of the predefined profiles shown in Table 3-8. (This, however, can also be accomplished by a thin wrapper, which is exactly what the command line `sandbox-exec(1)` is used for, along with the `–n` switch and a profile name).

TABLE 3-8: Predefined Sandbox Profiles

KSBXPROFILE CONSTANT	PROFILE NAME (FOR `sandbox-exec –n`)	PROHIBITS
`NoInternet`	`no-internet`	`AF_INET/AF_INET6` sockets
`NoNetwork`	`no-network`	`socket(2)` call
`NoWrite`	`no-write`	File system write operations
`NoWriteExceptTemporary`	`no-write-except-temporary`	File system write operations except temporary directories
`PureComputation`	`pure-computation`	Most system calls

The `sandbox_init(3)` function in turn, calls the `mac_execve` system call (#380), and the profile corresponds to a MAC label, as discussed earlier in this chapter. The profile imposes a set of predefined restrictions on the process, and any attempt to bypass these restrictions results in an error at the system-call level (usually a return code of `–EPERM`). The seatbelt may well have been renamed to "quicksand," instead, because once a sandbox is entered, there is no way out. The benefit of a tight sandbox is that a user can run an untrusted application in a sandbox with no fear of hidden malware succeeding in doing anything insidious (or anything at all, really), outside the confines of the defined profile. The predefined profiles serve only as a point of departure, and profiles can be created on a per-application basis.

Apple has recently announced a requirement for all Mac Store apps to be sandboxed, so the "voluntary" nature of sandboxing will soon become "mandatory," by the time this book goes to print. Because it still requires a library call in the sandboxed program, averting the sandbox remains a trivial manner — by either hooking `sandbox_init(3)` prior to executing the process[12] or not calling it at all. Neither or these are really a weakness, however. From Apple's perspective, the user likely has no incentive to do the former, because the sandbox only serves to enhance his or her security. The developer might very well be tempted to do the latter, yet Apple's review process will likely ensure that all submitted apps willingly accept the shackles in return for a much-coveted spot in the Mac store.

Controlling the Sandbox

In addition to the built-in profiles, it is possible to specify custom profiles in `.sb` files. These files are written in the sandbox's Scheme-like dialect. The files specify which actions to be allowed or denied, and are compiled at load-time by `libSandbox.dylib`, which contains an embedded TinySCHEME library.

DX-5562.053

You can find plenty of examples in `/usr/share/sandbox` and `/System/Library/Sandbox/Profiles` (or by searching for `*.sb` files). A full explanation of the syntax is beyond the scope of this book Listing 3-6, however, serves to demonstrate the key aspects of the syntax by annotating a sample profile.

LISTING 3-6: A sample custom sandbox profile, annotated

```
(version 1)
(deny default)              ; deny by default – least privilege
(import "system.sb")        ; include another profile as a point of departure


(allow file-read*)          ; Allow all file read operations
(allow network-outbound)    ; Allow outgoing network connections
(allow sysctl-read)
(allow system-fsctl)
(allow distributed-notification-post)

(allow appleevent-send (appleevent-destination "com.apple.systempreferences"))

(allow ipc-posix-shm system-audit system-sched mach-task-name process-fork process-exec)

(allow iokit-open           ; Allow the following I/O Kit calls
       (iokit-connection "IOAccelerator")
       (iokit-user-client-class "RootDomainUserClient")
       (iokit-user-client-class "IOAccelerationUserClient")
       (iokit-user-client-class "IOHIDParamUserClient")
       (iokit-user-client-class "IOFramebufferSharedUserClient")
       (iokit-user-client-class "AppleGraphicsControlClient")
       (iokit-user-client-class "AGPMClient"))
)

allow file-write*           ; Allow write operations, but only to the following path:
       (subpath "/private/tmp")
       (subpath (param "_USER_TEMP"))
)

(allow mach-lookup          ; Allow access to the following Mach services
       (global-name "com.apple.CoreServices.coreservicesd")
)
```

If a trace directive is used, the user-mode daemon `sandboxd(8)` will generate rules, allowing the operations requested by the sandboxed application. A tool called `sandbox-simplify(1)` may then be used in order to coalesce rules, and simplify the generated profile.

Entitlements: Making the Sandbox Tighter Still

The sandbox mechanism is undoubtedly a strong one, and far ahead of similar mechanisms in other operating systems. It is not, however, infallible. The "black list" approach of blocking known dangerous operations is only as effective as the list is restrictive. As an example, consider that in November 2011 researchers from Core Labs demonstrated that, while Lion's `kSBXProfileNoNetwork` indeed restricts network access, it does not restrict AppleEvents.[13] What follows is that a malicious app can trigger AppleScript and connect to the network via a non-sandboxed proxy process.

The sandbox, therefore, has been revamped in Lion, and will likely be improved still in Mountain Lion, where it has been rebranded as "GateKeeper" and is a combination of an already-existing mechanism: HFS+'s quarantine, with a "white list" approach (that is, disallowing all but that which is known to be safe) that aims to deprecate the "black list" of the current sandboxing mechanism. Specifically, applications downloaded will have the "quarantine" extended attribute set, which is responsible for the familiar "…is an application downloaded from the Internet" warning box, as before. This time, though, the application's code signature will be checked for the publisher's identity as well as any potential tampering and known reported malware.

Containers in Lion

Lion introduces a new command line, `asctl(1)`, which enables finer tuning of the sandbox mechanism. This utility enables you to launch applications and trace their sandbox activity, building a profile according to the application requirements. It also enables to establish a "container" for an application, especially those from the Mac Store. The containers are per-application folders stored in the `Library/Containers` directory. This is shown in the next experiment.

It is more than likely that Mac Store applications will, sooner or later, only be allowed to execute according to specific *entitlements*, as is already the case in iOS. Entitlements are very similar in concept to the declarative permission mechanism used in .NET and Java (which also forms the basis for Android's Dalvik security). The entitlements are really nothing more than property lists. In Lion (as the following experiment illustrates) the entitlements are part of the container's plist.

Experiment: Viewing Application Containers in Lion

If you have downloaded an app from the Mac Store, you can see that a container for it has likely been created in your `Library/Containers/` directory. Even if you have not, two apps already thus contained are Apple's own Preview and TextEdit, as shown in Output 3-8:

OUTPUT 3-8: Viewing the container of TextEdit, one of Apple's applications

```
morpheus@Minion (~)$ asctl container path TextEdit
~/Library/Containers/com.apple.TextEdit
morpheus@Minion (~)$ cd Library/Containers
morpheus@Minion (~/Library/Containers)$ ls
com.apple.Preview       com.apple.TextEdit
morpheus@Minion (~/Library/Containers)$ cd com.apple.TextEdit
morpheus@Minion (~/…Edit)$ find .
./Container.plist
./Data
./Data/.CFUserTextEncoding
./Data/Desktop
./Data/Documents
./Data/Downloads
./Data/Library
...
./Data/Library/Preferences
...
./Data/Library/Saved Application State
./Data/Library/Saved Application State
```

DX-5562.055

```
./Data/Library/Saved Application State/com.apple.TextEdit.savedState
./Data/Library/Saved Application State/com.apple.TextEdit.savedState/data.data
./Data/Library/Saved Application State/com.apple.TextEdit.savedState/window_1.data
./Data/Library/Saved Application State/com.apple.TextEdit.savedState/windows.plist
./Data/Library/Sounds
./Data/Library/Spelling
./Data/Movies
./Data/Music
./Data/Pictures
```

The `Data/` folder of the container forms a jail for the app, in the same way that iOS apps are limited to their own directory. If global files are necessary for the application to function, it is a simple matter to create hard or soft links for them. The various preferences files, for example, are symbolic links, and the files in `Saved Application State/` (which back Lion's Resume feature for apps) are hard links to files in `~/Library/Saved Application State`.

The key file in any container is the `Container.plist`, This is a property list file, though in binary format. Using `plutil(1)` to convert it to XML will reveal its contents, as shown in Output 3-9:

OUTPUT 3-9: Displaying the container.plist of TextEdit

```
morpheus@Minion (~/Library/Containers)$ cp com.apple.TextEdit/Container.plist /tmp
morpheus@Minion (~/Library/Containers)$ cd /tmp
morpheus@Minion (/tmp)$ plutil -convert xml1 Container.plist
morpheus@Minion (/tmp)$ more !$
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Identity</key>
        <array>
                <data>
                +t4MAAAAADAAAAABAAAABgAAAAIAAAASY29tLmFwcGxlLlRleHRFZGl0AAAA
                AAAD
                </data>
        </array>
        <key>SandboxProfileData</key>
        <data>
        AAD5AAwA9wD2APIA9wD3APcA9wDxAPEA8ADkAPEAjgCMAPgAiwDxAPEAfwB/AHsAfwB/
        AH8AfwB/AH8AfwB/AHoAeQD3AHgA9wD3AGsAaQD3APcA9wD4APcA9wD3APcA9wD3APgA
         ...  Base64 encoded compiled profile data ...
        AAACAAAALwAAAC8=
        </data>
        <key>SandboxProfileDataValidationInfo</key>
        <dict>
                <key>SandboxProfileDataValidationEntitlementsKey</key>
                <dict>
                        <key>com.apple.security.app-protection</key>
                        <true/>
                        <key>com.apple.security.app-sandbox</key>
                        <true/>
```

continues

DX-5562.056

OUTPUT 3-9 *(continued)*

```
                        <key>com.apple.security.documents.user-selected.read-write</key>
                        <true/>
                        <key>com.apple.security.files.user-selected.read-write</key>
                        <true/>
                        <key>com.apple.security.print</key>
                        <true/>
                </dict>
                <key>SandboxProfileDataValidationParametersKey</key>
                <dict>
                        <key>_HOME</key>
                        <string>/Users/morpheus</string>
                        <key>_USER</key>
                        <string>morpheus</string>
                        <key>application_bundle</key>
                        <string>/Applications/TextEdit.app</string>
                        <key>application_bundle_id</key>
                        <string>com.apple.TextEdit</string>
                          ...
                </dict>
                <key>SandboxProfileDataValidationSnippetDictionariesKey</key>
                <array>
                       <dict>
                               <key>AppSandboxProfileSnippetModificationDateKey</key>
                               <date>2012-02-06T15:50:18Z</date>
                               <key>AppSandboxProfileSnippetPathKey</key>
                        <string>/System/Library/Sandbox/Profiles/application.sb</string>
                       </dict>
                </array>
                <key>SandboxProfileDataValidationVersionKey</key>
                <integer>1</integer>
        </dict>
        <key>Version</key>
        <integer>24</integer>
</dict>
</plist>
```

The property list shown above has been edited for readability. It contains two key entries:

- **`SandboxProfileData`**: The compiled profile data. Since the output of the compilation is binary, the data is encoded as Base64.
- **`SandboxProfileDataValidationEntitlementsKey`**: Specifying a dictionary of entitlements this application has been granted. Apple currently lists about 30 entitlements, but this list is only likely to grow as the sandbox containers are adopted by more developers.

Mountain Lion's version of the `asctl(1)` command contains a `diagnose` subcommand, which can be used to trace the sandbox mechanism. This functionality wraps other diagnostic commands — `/usr/libexec/AppSandBox/container_check.rb` (a Ruby script), and `codesign(1)` with the `--display` and `--verify` arguments. Although Lion does not contain the subcommand, these commands may be invoked directly, as shown in Output 3-10:

DX-5562.057

OUTPUT 3-10: Using codesign(1) --display directly on TextEdit:

```
morpheus@Minion (~)$ codesign --display --verbose=99 --entitlements=:-        \
/Applications/TextEdit.app
Executable=/Applications/TextEdit.app/Contents/MacOS/TextEdit
Identifier=com.apple.TextEdit
Format=bundle with Mach-O universal (i386 x86_64)
CodeDirectory v=20100 size=987 flags=0x0(none) hashes=41+5 location=embedded
Hash type=sha1 size=20
CDHash=7b9b2669bddfaf01291478baafd93a72c61eee99
Signature size=4064
Authority=Software Signing
Authority=Apple Code Signing Certification Authority
Authority=Apple Root CA
Info.plist entries=30
Sealed Resources rules=11 files=10

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>com.apple.security.app-sandbox</key>
        <true/>
        <key>com.apple.security.files.user-selected.read-write</key>
        <true/>
        <key>com.apple.security.print</key>
        <true/>
        <key>com.apple.security.app-protection</key>
        <true/>
        <key>com.apple.security.documents.user-selected.read-write</key>
        <true/>
</dict>
</plist>
```

Entitlements in iOS

In iOS, the entitlement plists are embedded directly into the application binaries and digitally signed by Apple. Listing 3-7 shows a sample entitlement from iOS's `debugserver`, which is part of the SDK's Developer Disk Image:

LISTING 3-7: A sample entitlements.plist for iOS's debugserver

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
      <key>com.apple.springboard.debugapplications</key>
      <true/>
      <key>get-task-allow</key>
      <true/>
      <key>task_for_pid-allow</key>
      <true/>
```

continues

LISTING 3-7 *(continued)*

```
        <key>run-unsigned-code</key>
        <true/>
</dict>
</plist>
```

The entitlements shown in the listing are among the most powerful in iOS. The task-related ones allow low-level access to the Mach task, which is the low-level kernel primitive underlying the BSD processes. As Chapter 10 shows, obtaining a task port is equivalent to owning the task, from its virtual memory down to its last descriptor. Another important entitlement is `dynamic-codesigning`, which enables code generation on the fly (and creating `rwx` memory pages), currently known to be granted only to MobileSafari.

Apple doesn't document the iOS entitlements (and isn't likely to do so in the near future, at least those which pertain to their own system services), but fortunately the embedded plists remain unencrypted (at least, until the time of this writing). Using `cat(1)` on key iOS binaries and apps (like MobileMail, MobileSafari, MobilePhone, and others) will display, towards the end of the output, the entitlements they use. For example, consider Listing 3-8, which shows the embedded plist in MobileSafari:

LISTING 3-8: using cat(1) to display the embedded entitlement plist in MobileSafari

```
root@podicum (/)# cat -tv /Applications/MobileSafari.app/MobileSafari | tail -31 | more
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
^I<key>com.apple.coreaudio.allow-amr-decode</key>
^I<true/>
^I<key>com.apple.coremedia.allow-protected-content-playback</key>
^I<true/>
^I<key>com.apple.managedconfiguration.profiled-access</key>
^I<true/>
^I<key>com.apple.springboard.opensensitiveurl</key>
^I<true/>
^I<key>dynamic-codesigning</key>       <!-- Required for Safari's Javascript engine !-->
^I<true/>
^I<key>keychain-access-groups</key>
^I<array>
^I^I<string>com.apple.cfnetwork</string>
^I^I<string>com.apple.identities</string>
^I^I<string>com.apple.mobilesafari</string>
^I^I<string>com.apple.certificates</string>
^I</array>
^I<key>platform-application</key>
^I<true/>
^I<key>seatbelt-profiles</key>
^I<array>
^I^I<string>MobileSafari</string> <!-- Safari has its own seatbelt/sandbox profile !-->
^I</array>
^I<key>vm-pressure-level</key>
^I<true/>
</dict>
</plist>
```

iOS developers can only embed entitlements allowed by Apple as part of their developer license. The allowed entitles are themselves, embedded into the developer's own certificate. Applications uploaded to the App Store have the entitlements embedded in them, so verifying application security in this way is a trivial matter for Apple. More than likely, this will be the case going forward for OS X, though at the time of this writing, this remains an educated guess.

Enforcing the Sandbox

Behind the scenes, XNU puts a lot of effort into maintaining the sandboxed environment. Enforcement in user mode is hardly an option due to the many hooking and interposing methods possible. The BSD MAC layer (described earlier) is the mechanism by which both sandbox and entitlements work. If a policy applies for the specific process, it is the responsibility of the MAC layer to callout to any one of the policy modules (i.e. specialized kernel extensions). The main kernel extension responsible for the sandbox is `sandbox.kext`, common to both OS X and iOS. A second kernel extension unique to iOS, `AppleMobileFileIntegrity` (affectionately known as AMFI), enforces entitlements and code signing (and is a cause for ceaseless headaches to jailbreakers everywhere). As noted, the sandbox also has a dedicated daemon, `/usr/libexec/sandboxd`, which runs in user mode to provide tracing and helper services to the kernel extension, and is started on demand (as you can verify if you use `sandbox-exec(1)` to run a process). In iOS, AMFI also has its own helper daemon, `/usr/libexec/amfid`. The OS X architecture is displayed in Figure 3-2.

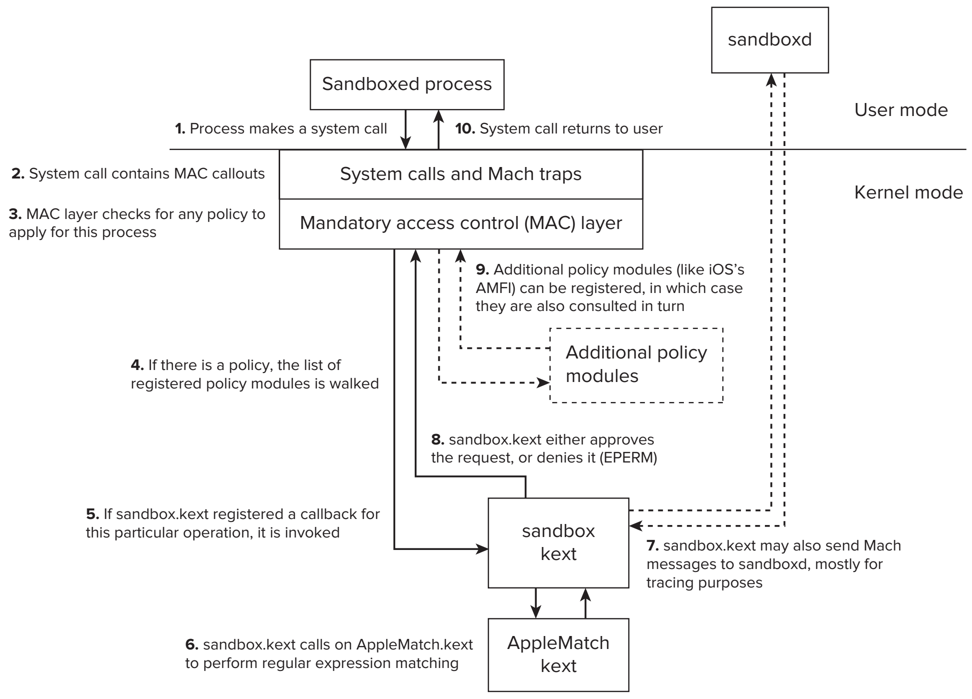

sandboxd
Sandboxed process
User mode
1. Process makes a system call
10. System call returns to user
System calls and Mach traps
Mandatory access control (MAC) layer
Kernel mode
2. System call contains MAC callouts
3. MAC layer checks for any policy to apply for this process
9. Additional policy modules (like iOS's AMFI) can be registered, in which case they are also consulted in turn
Additional policy modules
4. If there is a policy, the list of registered policy modules is walked
8. sandbox.kext either approves the request, or denies it (EPERM)
5. If sandbox.kext registered a callback for this particular operation, it is invoked
sandbox kext
7. sandbox.kext may also send Mach messages to sandboxd, mostly for tracing purposes
6. sandbox.kext calls on AppleMatch.kext to perform regular expression matching
AppleMatch kext

FIGURE 3-2: The sandbox architecture

DX-5562.060

Chapter 14 discusses the MAC layer in depth from the kernel perspective, and elaborates more on the enforcement of its policies, by both the sandbox and AMFI.

SUMMARY

This chapter gave a programmatic tour of the APIs that are idiosyncratic to Apple. These are specific APIs, either at the library or system-call level, providing the extra edge in OS X and iOS. From the features adopted from BSD, like `sysctl` and `kqueue`, OpenBSM and MAC, through file-system events and notifications, to the powerful and unparalleled automation of AppleEvents. This chapter finally discussed the security architecture of OS X and iOS from the user's perspective, explaining the importance of code signing, and highlighting the use the BSD MAC layer as the foundation for the Apple-proprietary technologies of sandboxing and entitlements.

The next chapters delve deeper into the system calls and libraries, and focus on process internals and using specific APIs for debugging.

REFERENCES

[1] "The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0," `http://www.trustedbsd.org/trustedbsd-usenix2003freenix.pdf`

[2] Apple Developer. "Sample Code — Reachability," `http://developer.apple.com/library/ios/#samplecode/Reachability/Introduction/Intro.html`

[3] Apple Technical Note 26117. "Mac OS X Server – The System Log," `http://support.apple.com/kb/TA26117`

[4] Sanderson and Rosenthal. Learn AppleScript: The Comprehensive Guide to Scripting and Automation on Mac OS X (3E), (New York: APress, 2010).

[5] Munro, Mark Conway. *AppleScript* (*Developer Reference*), (New York: Wiley, 2010).

[6] Apple Developer. "File System Events Programming Guide," `http://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/FSEvents_ProgGuide/`

[7] `http://fernlightning.com/doku.php?id=software%3afseventer%3astart`

[8] Apple Developer. "Code Signing Guide," `https://developer.apple.com/library/mac/#documentation/Security/Conceptual/CodeSigningGuide/`

[9] Technical Note 2250. "iOS Code Signing Setup, Process, and Troubleshooting," `http://developer.apple.com/library/ios/#technotes/tn2250/_index.html`

[10] "Charlie Miller Circumvents Code Signing For iOS Apps," `http://apple.slashdot.org/story/11/11/07/2029219/charlie-miller-circumvents-code-signing-for-ios-apps`

[11] Blazakis, Dionysus. "The Apple SandBox," `http://www.semantiscope.com/research/BHDC2011/`

[12] `https://github.com/axelexic/SanboxInterposed`

[13] Core Labs Security. "CORE-2011-09: Apple OS X Sandbox Predefined Profiles Bypass," `http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=advisory&name=CORE-2011-0919`

DX-5562.061